OpenConflict: Preventing Real Time Map Hacks in Online Games

Elie Bursztein
Stanford University
elie@cs.stanford.edu

Mike Hamburg
Stanford University
mhamburg@cs.stanford.edu

Jocelyn Lagarenne
Stanford University
jlagaren@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

***Keywords*-multi-player games, map hacks**

***Abstract*—We present a generic tool, *Kartograph*, that lifts the fog of war in online real-time strategy games by snooping on the memory used by the game. *Kartograph* is passive and cannot be detected remotely. Motivated by these passive attacks, we present secure protocols for distributing game state among players so that each client only has data it is allowed to see. Our system, OpenConflict, runs real-time games with distributed state. To support our claim that OpenConflict is sufficiently fast for real-time strategy games, we show the results of an extensive study of 1000 replays of Starcraft II games between expert players. At the peak of a typical game, OpenConflict needs only 22 milliseconds on one CPU core each time state is synchronized.**

I. Introduction

The online game industry is now among the largest entertainment industries in the world. According to the Entertainment Software Association[1], 67% of American households play video games, and computer games represent a 9.9 billion dollar market. Online gaming includes 64% of gamers. Among the many types of online games, real-time strategy (RTS) games are by far the most popular category, representing 35.5% of the PC game market (the second most popular category, first person shooters, only accounts for 10.1% of the market). Popular real-time strategy games include Starcraft II, Supreme Commander 2 and Age of Empires III.

The growing popularity of RTS games gave rise to global competitive video gaming. In 2010 the Electronic Sports World Cup attracted 500 professional players from around the world, and players could win up to $36,000 in cash prizes. Major tournaments, such as Major League Gaming (MLG), are held online with no direct supervision over competitors.

The competitive nature of the sport and the lack of direct supervision provides strong incentives to cheat. While active cheating, such as DDoS-ing an opponent, is easily detected and penalized, passive cheating is rampant. Cheating tools are often created by a laborious process of reverse engineering the client and changing it for the benefit of the player.

RTS games. In a typical real-time strategy game, player compete on a two-dimensional map divided into cells. A typical Starcraft II game, for example, is played on a map containing between 24000 and 36000 cells. Since RTS games are real-time (as opposed to turn-based), players compete on thinking speed as well as strategy. Each player gather resources in order to build structures and units as visible in Figure 1. As units move across the map, they encounter other players' units and may fight or collaborate with them. Most RTS games simulate a *fog of war* meaning that only certain parts of the map are visible at any moment (Figure 2). Areas of the map where the player has no units are hidden and the player cannot tell what is in those cells. The winner is the player who is able to destroy his opponent's base, or who achieves some other objective such as capturing flags. RTS games typically move quickly and finish in under an hour. Due to their fast pace, most RTS games are peer to peer: a central server may be used to set up the game, but the game itself is played over a direct network connection between the players. The lack of a central server to manage the game state makes it much harder to prevent client-side cheating.

Our contribution. We begin by presenting a generic attack tool, *Kartograph*, that among other things enables players to view the entire map by passively lifting the fog of war. *Kartograph* exploits the fact that RTS games store the entire game state on every player's computer, but only display information that players are allowed to see. *Kartograph* is a semi-automated tool that peeks into the game's memory and quickly discovers the game's internal memory layout. Once the game's memory layout is known, *Kartograph* extracts all information about the opponents and shows it to the cheating player. We emphasize the *Kartograph* is a generic tool that operates with no prior information about the game. We tested *Kartograph* against many popular RTS games, and were able to quickly lift the fog of war in all of them. This attack, called *map hacking*, is completely passive and cannot be detected by a remote observer.

Motivated by these attacks, we set out to design a generic defense against such passive attacks in RTS games. Our goal is to distribute the state of the game among the players so that each machine has only the part of the state it is allowed to know, and yet jointly the game proceeds as before. In theory, this can be done using a generic cryptographic protocol for *secure multiparty computation* [12]: on every state update, the players engage in a cryptographic protocol that implements

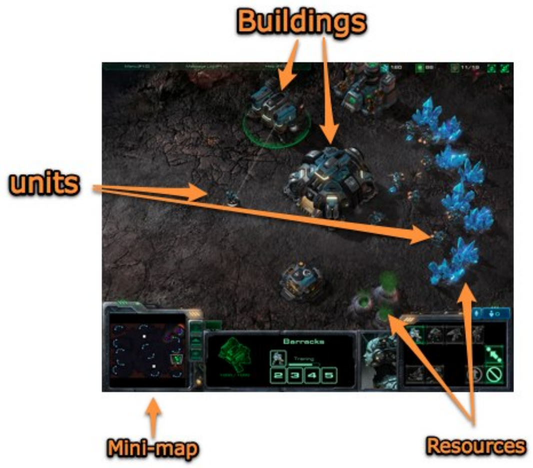

Figure 1. Structure of Real Time Strategy game

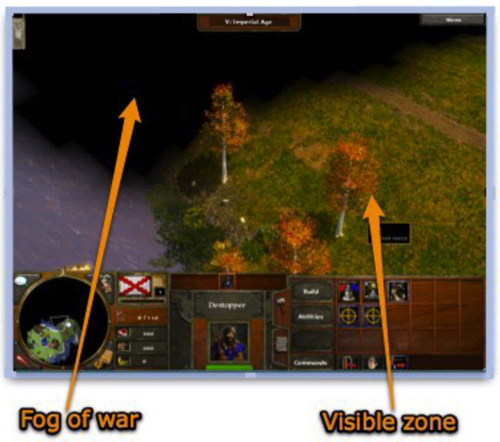

Figure 2. The zone that players can't see is called the fog of war

the game's rules.

At the end of the protocol every player has part of the state it is allowed to see and knows nothing else about the rest of the game. Unfortunately, since state updates in RTS games happen dozens of times per second, generic protocols for secure multiparty computation are far too slow for this purpose. Instead, we show that many RTS games can be efficiently implemented with distributed state using a suitably optimized protocol for *private set intersection* [10]. This approach virtually eliminates *all* information leakage about unit locations that can be gleaned from map-hacking.

To determine the performance requirements for running RTS games with distributed state we analyzed *1000* Starcraft II top player games and present the results in Section VI. The analysis shows, among other things, the board size, number of units, and number of actions per second for top players. In Section VIII we discuss the performance of our approach in light of the analysis of top Starcraft II games. We show that distributing state in RTS games is currently feasible, at least when the players have moderately high-end machines.

II. Background

Cheating in real time strategy games typically falls into three categories: abusing the *resource system*, *tampering with units* and *tampering with the map visibility*.

Abusing the resource system. Hacking the resource system gives the cheater more resources. The simplest approach is to find the location of resource values in memory and either freeze or increase them. Many games protect against this by obfuscating these values. For instance, Age of Empires III xors resource values with a "secret key."
Other ways to cheat include reducing the price of units by changing their cost in memory and having units gather resources more quickly.

Hacking the unit list. Hacking the unit list allows the attacker to overpower his opponent with units that are stronger, faster or tougher than ordinary units. This type of attack is often done by tampering with the unit's baseline statistics. In a peer-to-peer game this can detected by the opponent since the game state becomes inconsistent among the players. Another approach is to build units more quickly by tampering with unit build times, in order to overwhelm the opponent with a bigger army.

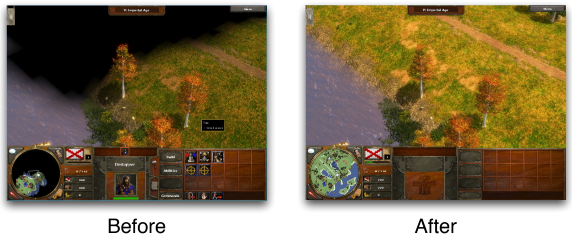

Figure 3. Map hacking Age of Empires 3 using *Kartograph*

Tampering with the map visibility. The last type of cheating involves tampering with the visibility restrictions enforced by the game. This type of attack, known as map hacking (figure 3), is the hardest to detect because it is fully passive. Of these three attacks, map hacking is also the hardest to perform because it requires a deep understanding of how the game works, and in particular how map information is stored. Map hacking is frequently performed by injecting at run-time a DLL that overrides the functions responsible for enforcing the fog of war. Anecdotally, a creative map hack for Warhammer II fooled the game into running in replay mode, causing it to display the entire map to the cheating player.

III. A Generic Tool for Map Hacking

In this section we describe our *Adversarial Game Instrumentation* (AGI) techniques that we use to perform memory attacks on virtually every game. We implemented these techniques in a tool called *Kartograph* written in C#. *Kartograph* only runs on 64-bit Windows because it needs more than 4 GB of memory to analyze modern games.

Before describing how *Kartograph* works, we first review why the opposing player's information is always present in memory. In a peer-to-peer game, the easiest way to keep game data in sync between players is for each client to broadcast its entire state (units, buildings, etc) to all other game clients. We call this the *push approach*. As far as we know, this is the only approach used in practice because it minimizes latency and code complexity. However, the opponents' game clients must be trusted to enforce visibility restrictions, and as we will see they may not always deserve that trust. An alternative approach, the *pull approach*, has players request their opponents' data only as needed. This approach adds complexity and network latency, and when applied straightforwardly its security benefits are limited: the requests themselves still leak enough information to give a cheater a considerable advantage. We discuss how to mitigate this information leak through cryptography in section VII.

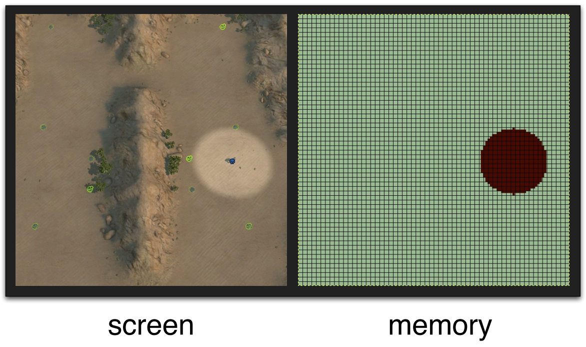

Figure 4. Memory representation of a visibility map structure

As explained earlier, memory attacks manipulate the game's memory structures rather than changing its data files. To perform a memory-based map hack, the hacker first needs to find the memory structures used to render the map, as shown in figure 5. In the easiest case all the information needed is stored in a single 2D array called the *visibility map* (figure 4). In practice, however, many games store their map data in multiple memory structures, which makes the attack slightly more complicated. However, the same techniques still apply, so we will assume for illustrative purposes that the goal is to find and reverse-engineer a single visibility map and a single unit list.

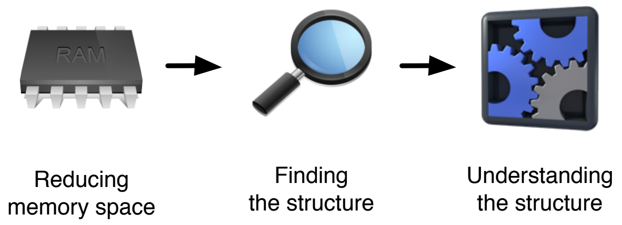

Figure 5. Memory-based hacking overview

Our approach to memory map hacking has three steps summarized in figure 5. First, we need to narrow down the memory range in which we will search for the map data. Narrowing down the memory location of the map is mandatory because the game memory is too vast to explore. For example, Starcraft II uses 850 MB of memory and Supreme Commander 2 uses over 1 GB (see table II). Second, once the search space is reduced, we need to identify the map structures. Finally we need to understand how these structures work in order to extract the relevant information. Past approaches would use a debugger and a decompiler for each step, but this is very tedious. Furthermore, using a debugger on the game is likely to trip its security mechanisms such as the Blizzard Warden [26]. Instead, we simply snoop on the game's memory while playing it in a manner that will leak the information we need. We call this technique *Adversarial Game Instrumentation* as we exploit game functionalities to leak the information we need. We now describe in detail how we perform each of the three steps.

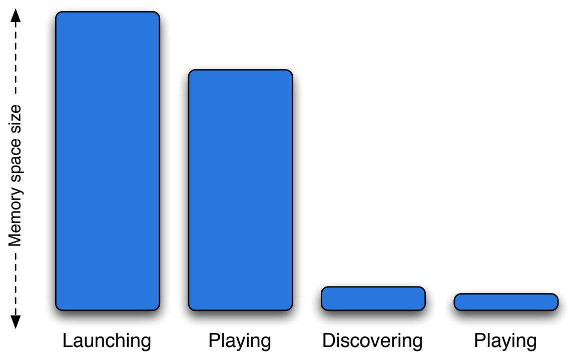

Figure 6. Reducing memory space via *Adversarial Game Instrumentation*

Reducing the memory space. In general, the memory that contains unit positions only changes when units move, and likewise the visibility map changes only when the visible region changes. To force the game to leak the location of the visibility map, we perform the four-step procedure in Figure 6.

In the first step, we launch the game and read all the memory space where the map might possibly be. This includes all the memory pages of the process's main module that are marked as $ReadWrite$, $Commit$ and $Private$: $ReadWrite$ because visibility changes during the game, $Commit$ because the map has changed since being allocated, and $Private$ because visibility is not mapped from a file. Second, we move the camera around and trigger as many actions as we can without discovering any new parts of the map. The goal is to change as much of the game memory as possible, other than the map data. After a minute or so, we ask *Kartograph* to eliminate all the memory blocks that changed since launch time. Third, we discover a portion of the map by using a unit to "scout" an unknown area. Because we are going to use visualization techniques to find the map, we need to create a "nonlinear" scouting pattern, like the one visible in figure 13, that is easy to spot. This time we ask *Kartograph* to keep only the memory blocks that changed. Because the game's memory contains a large amount of constant data, this step gives the biggest reduction in search space. Table II shows that in practice this step is able to reduces the memory space by as much as a factor of 250. Finally, we again play without changing the map and ask *Kartograph* to remove the pages that have changed. While this step might appear redundant with the second step, in practice it helps considerably. In particular, it consistently halves the search space in Supreme Commander 2 (see table II).

Finding the visibility map. Once we have narrowed down the potential locations of the visibility map to a manageable size, we find the map by looking for our scouting pattern in memory. We do so by generating a *heat map* representation of memory, as shown in Figure 7. Each color represents a different memory value. Black represents gaps in a compressed fashion, as they can be tens of megabytes wide. One difficulty of using the heat map visualization to find the map structure is that different games use different data types for the visibility map. It is often necessary to test bytes, shorts and ints separately. Therefore, we designed *Kartograph* to switch from one representation to another seamlessly. The other difficulty is that memory structures are not properly aligned and then appear skewed in the visualization as shown in Figure 8.

Nevertheless, the map data is quick and easy to find with minimal experience. For example, we used the heat map visualization to create a map hack for Age of Empires III in under 5 minutes while demoing *Kartograph* at the Defcon 2010. *Kartograph* makes this visualization technique easier by using frequency analysis that removes all the zones that

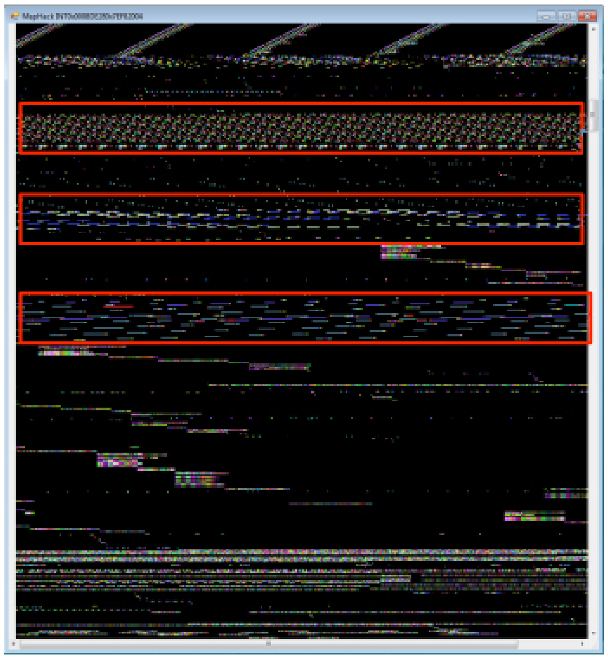

Figure 7. *Kartograph* heat map visualization of the reduced memory space. Possible map structures are circled in red.

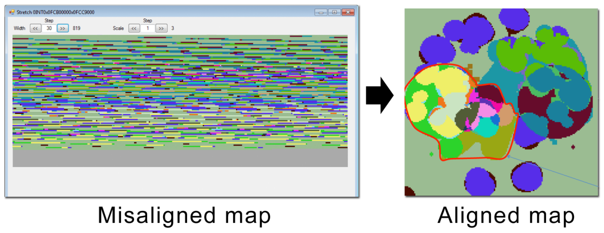

Figure 8. Example of a misaligned visibility map for Age of Empire III

do not contain repetitive data. While this technique works well for the visibility map, it is not efficient for finding unit structures, which are an order of magnitude smaller and have a less recognizable shapes. Accordingly, we use a different technique to locate the unit list, as described in the next section.

Understanding the visibility map. The final step to build a map hack is to understand how the visibility map works so we can extract and interpret the information contained in it. To understand how the structure works we perform what we call a diff-map analysis: First *Kartograph* takes a snapshot of the memory blocks that we believe to be the visibility map. Then we move a unit, and *Kartograph* generates a bi-color diff-map, where the red pixels represent the memory blocks that changed and the blue pixel represent the block that didn't change (Figure 9). If we are indeed looking at the visibility map, the diff-map will contain either a shape that represents the part of the map that the unit discovered (e.g.

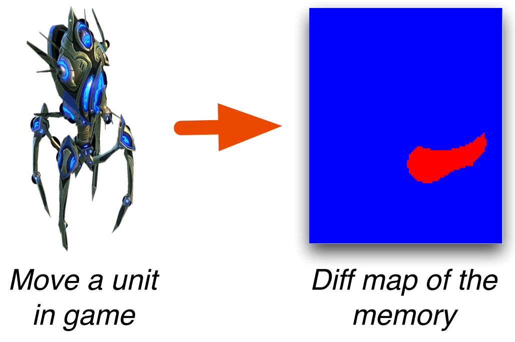

Figure 9. Diff-map visualization of adversarial game instrumentation

Age of Empires III – figure 13) or two shapes corresponding to the unit's previous and current position (e.g. Supreme Commander 2 – figure 14). Finally we try to understand the map's structure by looking at the values that have changed.

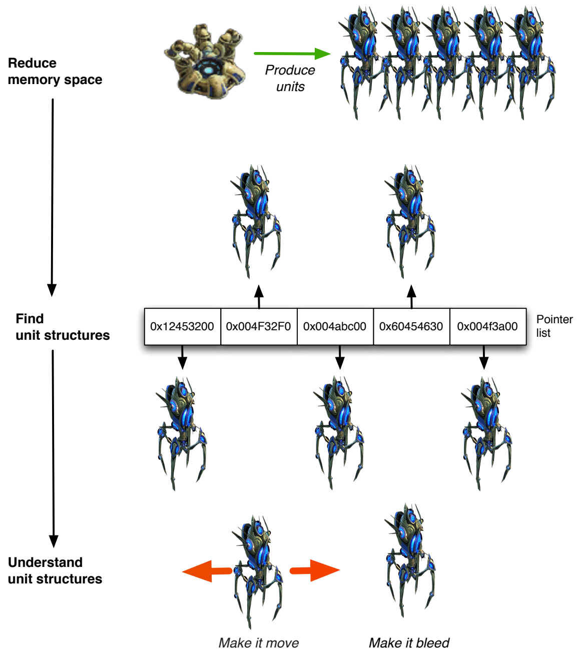

Figure 10. Finding unit structure using *Adversarial Game Instrumentation* and in-memory shape analysis

Finding the unit list. Finding the unit list is more challenging than finding a map because it is a much smaller (1.5KB vs. 500KB) and more complex structure. To find the unit list we use the procedure summarized in Figure 10. We build five units in a row and have *Kartograph* eliminates all memory blocks that violate the following criterion between each unit construction:

1) The memory block must have changed or a contiguous block must have been allocated. We must consider changing blocks as a valid candidates because in some games, such as Warcraft III and Starcraft II, the unit list is pre-allocated.
2) The newly allocated block must be larger than 64 bits or contain a valid pointer address. To test that a 64-bit value is a valid pointer, we perform a pointer dereferencing analysis and verify that the value points to a valid memory block with the correct flags (ReadWrite, Commit and Private).

Game	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5
Supreme Commander 2	176454	13546	428	55	12
Age of Empires 3	3443	177	48	29	10

Table I
NUMBER OF POSSIBLE LOCATIONS FOR THE UNIT LIST

Table I shows that in practice we can usually find the unit list after building 5 units, even if we start with a lot of candidates. Once we have isolated the unit list, *Kartograph* uses a pointer analysis to help visualize the list. Once again we analyze the result of precise game actions to understand how the unit's map coordinates, health, mana etc. are stored and obfuscated. For example, we move the unit to find its coordinates, damage it to find its health, and have it cast spells to find its mana. After several experiments, we have all the data we need to reverse-engineer the useful part of the unit's structure. If the opponents' units are stored separately, we can perform the same technique by playing against a friend.

Network analysis. To help us find the location of the opponents units in memory, *Kartograph* correlates memory changes with received packets. *Kartograph* intercepts and manipulates network packets using a layered service provider in the Winsock stack. A similar approach was used before by Levin [21].

Kartograph network engine's visualization is shown in figure 11. It first divides packets into buckets based on size, and then shows for each bucket a visualization that combines a heat-map and a diff-map. The heat map represents the packets stacked in the order of arrival, with the first packet at the bottom and the most recent on top. The width of the heat map represents offsets in bytes. The Figure 11 shows that if a given position holds a constant (e.g. a command id) then we observe a monochrome stripe. If it holds a counter, we see a gradient and if it is an arbitrary, random or encrypted value we see a scrambled list of colors. At the very top of the heat-map, *Kartograph* draws a diff-map that summarizes which offsets of the packet changed across time. The constant offsets are blue and the changing ones are red.

IV. GAME HACKING IN PRACTICE WITH *Kartograph*

In this section we describe some of the issues we came across while using *Kartograph* against popular games. We also discuss the effectiveness of our memory space reduction techniques on practical examples. We plan to release *Kartograph* at the same time as our cryptographic library used to defend against it. We analyzed many games, listed in table II: old games such as Command and Conquer Tiberian Sun (1999), more recent games such as Wacraft III (2003) and Age of Empires III (2005), and the most recent games such as Supreme Commander 2 (2010) and Starcraft II (2010).

Game	Launch	Play	Discover	Play more
Starcraft II	850M	725M	2M	1.3M
C&C Tiberium Sun	75M	73M	400K	400K
C&C Red Alert 2	101M	100M	935K	915K
C&C Red Alert 3	660M	635M	4.4M	1.6M
Age of Empire III	245M	243M	2.7M	2.5M
Supreme Commander 2	1.2G	629M	2.5M	1.5M
Civilization IV + ext	340M	293M	2M	1.9M
Anno 1701	432M	413M	1.9M	1.8M
Warcraft III	129M	124M	1.9M	1.8M

Table II
MAP SIZE REDUCTION

A. The visibility map structure

One of the most fascinating things when reverse-engineering a game is to uncover which data structures it uses. Each of the 15 games we analyzed had unique structures and data representations. In particular we analyzed the Command and Conquer (C&C) series to see if we could find a pattern, but it turned out that that these structures changed radically from one opus to another, probably due to big changes in the game engine. Overall we found that the representation of map information falls into two broad categories:

- a **bitmap** representation with an array that corresponds directly to map data, or
- a **composite** representation that stores map data in different structures and combines them during rendering.

In either case, as shown in table II, our reduction algorithm works well and quickly narrows down the map structures location. In practice narrowing down the map location can be accomplished under 2 minutes. There are two difficulties with this method. First, on modern games this approach requires a lot of memory because we need to snapshot the entire main module's memory. This requires twice its memory size because we need to store each memory address and each memory value. As a result, for games like Supreme Commander 3, we need 1.2 GB for the game and 2.4 GB for *Kartograph*. That is why *Kartograph* was designed to only work on 64-bit Windows. The second issue is the way the game allocates resources. In some cases, memory is allocated in 64 MB chunks, which causes the heat-map to take a very long time to render. The fact that games use so much memory makes real-time visualization of the memory very difficult. We are experimenting with real-time visualization techniques using Direct3D to speed up the visualization process.

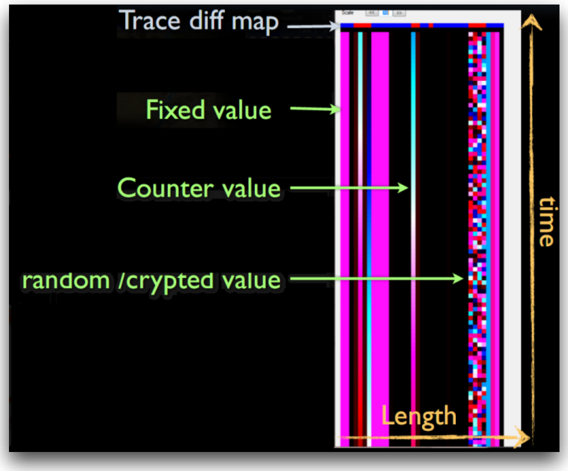

Figure 11. *Kartograph* network visualization and manipulation UI

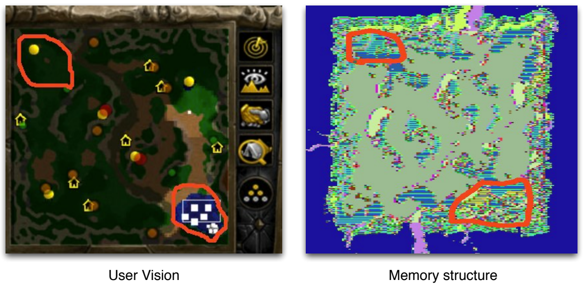

Figure 12. Heatmap view of the Warcraft 3 map structure

The bitmap representation. An example of the bitmap representation used by Warcraft III is shown in figure 12. It is easily seen that the information stored in the bitmap contains the opponents' buildings, which are filtered out before being displayed to the player.

Composite representations. In our experience, composite representations are more common than simple bitmap representations. These representations are more difficult, both because the map data is stored in multiple data structures, and because these structures vary wildly from one game to another. The following examples illustrate how diverse the data structure are:

- **Age of Empires III** As shown in figure 13, Age of Empires III uses separate structures to store resource

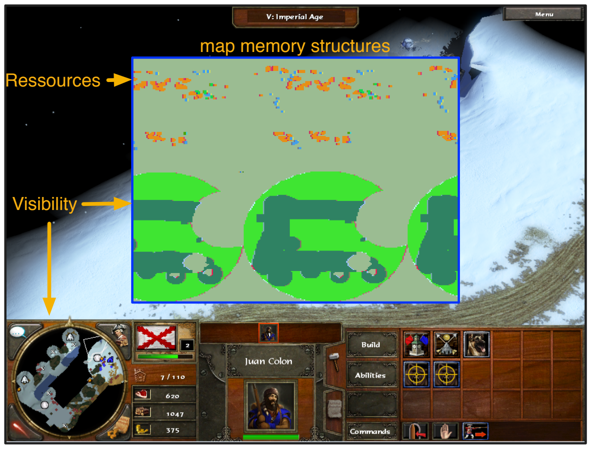

Figure 13. Heatmap view of some of the Age of Empire 3 map structures

information and visibility information. The information visible for each player is encoded using a bit fields.

- **Civilization IV** uses a filter-based approach: its visibility map contains for each cell a color that is applied by the game to create the fog of war. Each cell's color filter is encoded as a 32-bit integer, with its 8 most significant bits used for alpha. Unexplored cells contain a straight black mask (`0xFF000000`), and explored but currently unobserved cells are masked with a gray mask (`0xFF6D6D6D`).
- **Supreme Commander 2** encodes the visibility as a short integer, representing how many units are able to see a given cell of the map. An example of this cumulative visibility map is visible in figure 14.

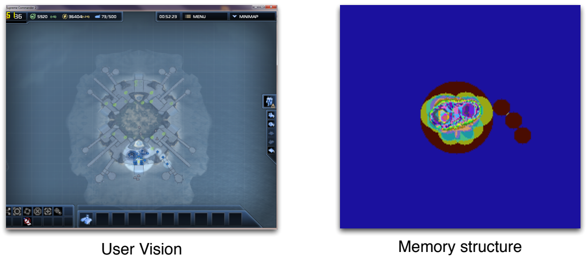

Figure 14. Heatmap view of the Supreme Commander 2 cumulative visibility map

B. Unit analysis

While the unit list is an order of magnitude smaller than the map, it is still possible to quickly narrow down its location as shown in Table I. As we find more units the number of possible locations for the unit list decreases rapidly. At first we expected to see units stored in a linked list, which is far from trivial to reverse-engineer. However, in practice most games use a stack, either with pointers to units (e.g. Warcraft III and Starcraft II), or with a pointer and a unit ID (e.g. Age of Empires III).

Consequently, instead of using the complex linked-list analysis algorithm we originally developed, we ended up using the simplified algorithm described in the previous section. The last point worth mentioning is that unit hit points are often obfuscated. This is done to prevent an attacker from searching memory for the hit point value shown on screen. However, our *Adversarial Game Instrumentation* technique nullifies this defense, allowing us to quickly reverse-engineer unit structures despite simple obfuscation.

C. Using the game as a map hack

Since we know the map structures' locations and format, we can take *Kartograph* one step further and trick the game into displaying the entire map and lifting the fog of war. For example, in Supreme Commander 2 we can lift off all the visibility restrictions by changing all the 0's in the cumulative game map to a positive number. Because with *Kartograph* we are able to precisely rewrite the map structure with a meaningful value, we can not only turn the game into a map hack but also create all sorts of strange effects. For instance, by re-writing only part of the Civilization visibility filter map, we can selectively reveal only a portion of the map, as shown in figure 15. During online play, some games check visibility map consistency. In this case, we must either rewrite network packets, or write the map hack as an external program that overlays itself on top of the game.

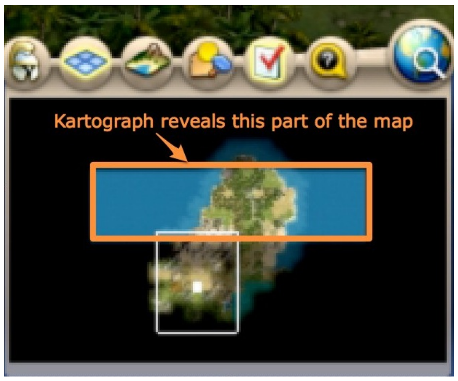

Figure 15. Partial rewrite of the Civilization 4 visibility filter map. The white rectangle is the allowed visibility region while the thick visibility rectangle was added by Kartograph.

V. Preventing passive Map Hacks

We now turn to defending against *Kartograph* and other passive information extraction attacks. We first define the passive eavesdropper threat model and then describe OpenConflict, our system that defends against such attacks. We discuss active attacks in section IX.

Some attacks in the previous section display information about the opponent by removing the fog of war mask from the game. While this kind of attack actively changes data in game memory, we still treat it as a passive attack. Indeed, removing the fog of war is simply a clever trick for showing the player data about the opponent. We don't change the game state or network traffic. We could have just as easily left the fog of war unchanged and presented this data in a side panel.

A. Threat model: passive eavesdropping adversaries

We assume the attacker (i.e. a cheating player) has complete control of his machine and can run arbitrary privileged software in parallel to the game. In particular, the attacker can run software that constantly takes memory snapshots of the game's internal memory. We assume that the attacker has a complete understanding of the game's memory layout and can identify and parse all data structures stored in memory. Armed with these tools, the attacker runs a program P that takes a memory snapshot every few milliseconds, parses all data structures and extracts information about the opponents' units. If successful this information is displayed to the attacker. We say that a passive attacker defeats the game if the attacker can write a program P that reveals information about the opponent beyond what is allowed by the game's rules. Otherwise we say that the game is secure against a passive adversary.

We assume that the game has a peer-to-peer architecture, as is the case for Starcraft II and most other fast paced online games. That is, each party communicates directly with the others without using a central server to manage the game state. In slower games like World of Warcraft, where a central server runs the game, security against passive attacks is straightforward since the server only tells each client precisely what the client is allowed to know. As we will see, security in fast peer-to-peer games is much harder.

B. Challenge

To build secure games we will make use of certain multi-party cryptographic protocols. Our challenge is to design sufficiently fast protocols so that the added game latency is imperceptible to the players.

VI. Case Study Starcraft II

Before describing our solution, we analyze the scale of the problem by measuring how many units and how many map cells each player sees in Starcraft II. To obtain these numbers we analyzed 1000 Starcraft II replays from top players. At the end of a Starcraft II game players have the option to save their game and re-watch it later. Replay files are commonly posted online for other players to comment and learn. Another source of replay files is tournaments like the *Global Starcraft II League* (GSL). As a result, there is a large pool of real Starcraft II replay files for us to analyze.

Methodology. Top players in games like Starcraft II tend to have a high number of actions per minute (APM). These high-APM games are a good stress test of our system. We selected 1000 replays with APM over 120 which is at the very high end of the replays available. While this corpus of games gives us statistics that are higher than for average players, we will show that our system handles these high-APM games with little latency.

Analyzing Replays. The replay file format, *.mpq*, is partially documented and is interpreted by the game engine as it replays the game. To mine these replay files we wrote a crude "game engine" that replays games. Our game engine uses the standard initial Starcraft II unit statistics, such as speed, visibility, range, and hit-points. Since the engine ignores collisions and does crude fight reconstruction, it outputs an over-approximation of the data generated during the game. For example, when in doubt it leaves units alive and over-estimates the number of cells visible to players. Consequently, while the results below are not 100% accurate, they represent a conservative estimate of the amount of data generated during the game.

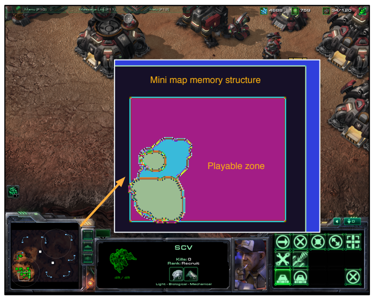

Figure 16. Starcraft II mini map memory structure heatmap visualization

Map size vs. playable size. Every replay file contains information about the map played. Two important data for us are the total number of cells in the map and the number of cells in the playable area of the map, namely where players can move their units. The difference between the two is obvious when the Starcraft II mini map memory structure is visualized with *Kartograph* (Figure 16).

For our 1000 games, the map size is between **24320** and **36864** cells and the playable size is between **15180** and **24640** cells. While these numbers are an upper bound on each player's visibility, we show below that in reality visibility is far less because the game mechanics favor players that keep their units in a small number of clusters.

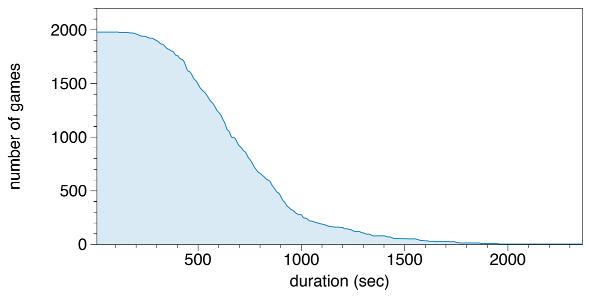

Figure 17. Starcraft II game duration cumulative value (10-second buckets)

Game duration. Figure 17 shows the cumulative number of game that last more than X seconds. After 6 minutes the number of games still in progress drops quickly and after 30 minutes (1800 seconds) most of the games are finished. An important consequence is that our solutions need only secure data for at most a few hours. For fast-paced games like Starcraft II, the value of the data quickly becomes worthless, so preserving secrecy for a few minutes is sufficient. Hence, there is no need to use traditional cryptographic strength and we can get away with using smaller parameters everywhere.

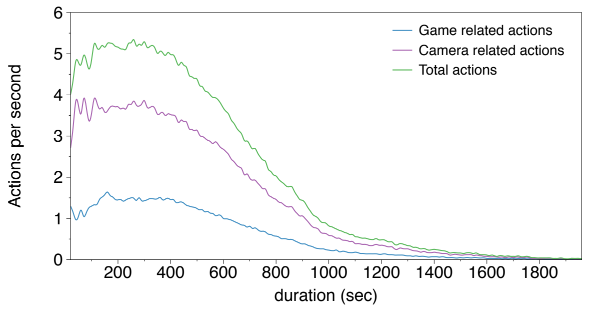

Figure 18. Starcraft II action per second breakdown (10-second buckets)

Actions per second. In order to fix accurate performance requirements, we measured the rate at which players act. Figure 18 shows that an average top player does about **1.6** game-related actions per second. The remaining actions are camera moves (changing view point) which have no impact on game state. The figure shows that as games draw to a close payers fatigue and mostly focus on marching to their opponents bases.

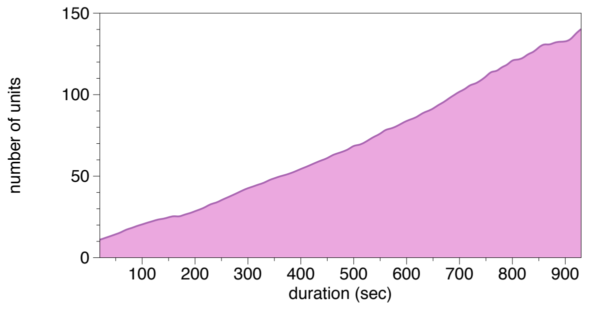

Figure 19. Starcraft II number of units by time (10-second buckets)

Visibility. As the game progresses players acquire more units. Figure 19 shows the number of units for one player as a function of time. Players typically never have more than **150** units, including buildings. We stopped plotting after 900 seconds since very few games last that long.

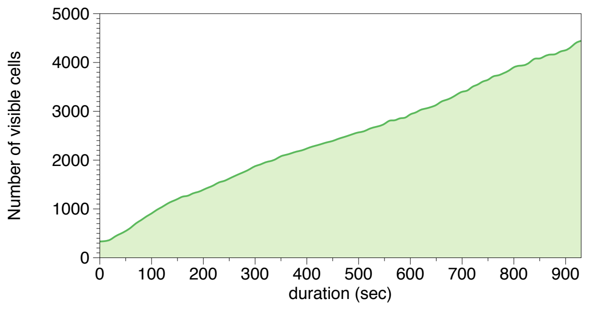

Figure 20. Starcraft II number of visible cells by time (10-second buckets)

Figure 20 shows an over-approximation of how many map cells are visible to each player as the game progresses. Since players tend to keep their units in a few clusters, the number of visible cells grows slower than the number of units. Players typically do not see more than **4500** cells at a given time, which is about **33%** of the total map. In computing these estimates, we ignored occlusion due to terrain height. We also ignored extra visibility from the game's Xel'Naga watchtowers, but as we see later, these towers actually decrease OpenConflict's workload.

VII. Defending against map hacking

We now describe our approach to prevent the passive map-hack attacks described in section III. Since the attacker (i.e. a malicious player) can peek into the game's memory on his machine, our goal is to convert the game into one where each player's machine only stores information that the player is authorized to see.

For each pair of opposing players (let's call them Alice and Bob), Alice can see some of Bob's units (those close to her own units), but not others. We want Bob to send Alice data on those units that she can see, but not on those units that she cannot see. The problem is that Bob doesn't know, and shouldn't be allowed to know, which of his units Alice can see. A solution to this problem is to use an oblivious intersection protocol.

A. Oblivious Set Intersection

Let M be the set of all cells on the map. Each cell may contain units, buildings and other objects; we will refer to these generically as "units". Alice has units scattered across the map and each unit has a visibility radius. Taking the union of all of Alice's visibility regions gives the set $V_A \subseteq M$ of cells that Alice can see. Let $U_B \subseteq M$ denote the set of map cells containing Bob's units.

The game needs to show Alice all the units belonging to any player within her visibility region. To get Bob's data, Alice's machine needs to determine $V_A \cap U_B$ subject to two constraints stated informally as:

1) Bob should learn nothing about V_A (so that he learns nothing about the location of Alice's units), and
2) Alice should learn nothing about U_B other than $V_A \cap U_B$ (so that she learns nothing about the location of Bob's units).

Given $V_A \cap U_B$ Alice's machine can correctly render the cells in V_A, as required. Likewise, we want Bob to learn $V_B \cap U_A$, namely Alice's units in Bob's visibility regions, and this will be accomplished by running the protocol in the reverse direction.

The problem of computing $V_A \cap U_B$ subject to the two constraints above is called **oblivious set intersection** and many protocols for this problem have been proposed [10], [20], [14], [5], [13], [17], [18], [8], as discussed in Section X. For each cell in the intersection, Alice wants to learn information about Bob's units in that cell. We encapsulate this as a function $f_B : U_B \to D$ for some data domain D. We would like a protocol whereby Alice learns the value of f_B on $V_A \cap U_B$ but not anything about $U_B \backslash V_A$, and Bob learns nothing about Alice's units.

Since we are mostly concerned with eavesdropping attacks we will assume a passive threat model (a.k.a. honest but curious) meaning that Alice and Bob will execute the protocol faithfully, but they wish to gain information by looking at the flows. Formally, the security requirement is that

1) Given U_B there is a simulator that simulates Bob's view of the protocol (and therefore Bob learns nothing from the interaction with Alice), and
2) Given V_A and $V_A \cap U_B$ there is a simulator that simulates Alice's view of the protocol (and therefore Alice learns nothing from the interaction other than $V_A \cap U_B$).

A protocol satisfying these two requirement is said to be private for passive adversaries.

B. Oblivious Function Evaluation

We found that a good starting point for our settings is an oblivious intersection protocol due to Jarecki and Liu [18] which uses **oblivious function evaluation** as a sub-protocol (a related protocol is presented in [8]). We describe the protocol while adapting and optimizing it to our settings.

Oblivious function evaluation is defined with respect to a keyed function $o_k(v)$ that uses a secret key k to map a value v to some domain. Bob holds the secret key k. An oblivious function evaluation protocol is a way for Alice to learn $\{o_k(v) : v \in V_A\}$, without learning $o_k(w)$ for any $w \notin V_A$, and without Bob learning anything about V_A.

For our function $o_k(v)$, we choose a group $\mathbb{G}$ of prime order q (in particular, a subgroup of the points on an elliptic curve) and a hash function $H_1 : M \to \mathbb{G}\backslash\{1\}$. Bob's key k is a random integer in $[1, q-1]$, and the function $o_k(\cdot)$ is defined as:

$$o_k(v) := H_1(v)^k \in \mathbb{G} \ .$$

Now, the oblivious function evaluation protocol runs as follows:

- Alice chooses a random integer $r \in [1, q-1]$ and sends $x := H_1(v)^r$ to Bob;
- Bob responds with $y := x^k = H_1(v)^{rk}$;
- Alice computes $o_k(v) = y^{r^{-1}}$.

Simple variants of this can be more efficient for certain groups $\mathbb{G}$. For example, Alice can blind v in step (1) as $H_1(v) \cdot g^r$ for some generator g. She unblinds y in step (3) as $y^{r^{-1}}/g^r$. This variant improves efficiency of step (1) since Alice's exponentiations are relative to a fixed base g.

Theorem 7.1: If the Computational Diffie-Hellman (CDH) assumption holds in $\mathbb{G}$ then the protocol above is a secure oblivious function evaluation protocol when H_1 is modeled as a random oracle.

Proof: Bob sees only one random group element from Alice, so his view of the interaction can be simulated by choosing a random element in $\mathbb{G}$. Hence, Bob learns nothing from the interaction. Now, suppose Alice and Bob run the protocol on V_A so that Alice learns $\{o_k(v) : v \in V_A\}$. Suppose Alice can find some $w \notin V_A$ so that she can compute $o_k(w)$. We show how to use Alice to solve a CDH challenge, which is compute h^k given g, g^k, h. We run a simulation with the random oracle rigged to give g^{s_v} on $v \in V_A$ and h^{s_w} on $w \in M\backslash V_A$, where the simulator knows the values s_v and s_w.

Then Alice's queries g^{rs_v} can be answered with g^{krs_v}, but if Alice computes $o_k(w) = h^{ks_w}$ then $h^k = o_k(w)^{s_w^{-1}}$ solves the CDH challenge. ■

This simple proof works only because Alice is honest. Jarecki and Liu show how to handle a dishonest Alice, but at the cost of a much stronger assumption than CDH [18].

C. The Basic Oblivious Set Intersection Protocol

To compute $V_A \cap U_B$ Alice and Bob now do the following, as shown in Figure 21:

1) Bob first chooses a random key $k \in \{1, \ldots, q-1\}$.
2) For each $u \in U_B$ Bob locally does the following:
   a) He computes a key k_u as $k_u \leftarrow H_2(o_k(u))$ where H_2 is a hash function.
   b) He encrypts the message $f_B(u)$ using the key k_u in a symmetric encryption scheme providing authenticated encryption (e.g. where ciphertexts include a MAC).

   Bob sends the list of resulting ciphertexts to Alice.

   Note that we must ensure that the *length* of the encryption of $f_B(u)$ not leak information about $f_B(u)$. Our system ensures this by breaking $f_B(u)$ into fixed sized chunks (possibly padding the last chunk) and encrypting each chunk separately. The chunks from all $|U_B|$ ciphertexts are then sent to Alice in a random permuted order.
3) Alice engages in the oblivious function evaluation protocol with Bob to obtain $y_v := o_k(v)$ for all $v \in V_A$. Note that Bob learns nothing about V_A.
4) For each y_v Alice computes $k_v \leftarrow H_2(y_v)$ and tests if one of the ciphertexts received from Bob decrypts correctly under y_v (i.e. the decryption algorithm does not return $\perp$). If so then $v \in V_A \cap U_B$ and she learns $f_B(v)$, as required.

D. Chaff

The basic protocol leaks to Bob the number of cells in Alice's visibility set V_A. It leaks to Alice the sum of the lengths of $f_B(u)$ for $u \in U_B$ which reveals information about the total number of units that Bob has. While Bob cannot completely hide the total number of his units, he can reduce the amount of information that Alice infers by adding a **chaff** in the form of random, meaningless data chunks. Alice can't tell the difference between these random chunks and units that she can't see, so she only knows an upper bound of the number of Bob's units. Since these chunks are random, they cost almost nothing other than the bandwidth they consume.

Conversely, Alice can hide the size of her visibility map by sending meaningless, random queries. Bob won't be able to tell the difference between these random queries and Alice's real visibility queries. However, Bob must respond to all queries, so the chaff increases his workload.

E. Hypergrids

Visibility regions are almost always large, continuous shapes rather than disconnected point sets. To take advantage of this property, we can construct multiple levels of grid cells ("hypergrids"), each one coarser than the last. For example, we could divide the grid into 1×1, 2×2, 4×4 and 8×8 tiles. Then Alice can decompose V_A into a union of grid cells and hypergrid cells in order to minimize computation and bandwidth consumption. That is, if Alice can see all the tiles in some hypergrid cell, she sends a query for the entire cell instead of for each of the tiles inside it. As a result, Alice will send a number of queries proportional to the perimeter of V_A rather than its area. Figure 22 shows how hypergrids reduce the number of elements in the visibility set by a factor of **6** during the game. The top line refers to the total number of visible cells. The bottom line shows that number of visible hypergrid cells where each tile contains a number of adjacent cells.

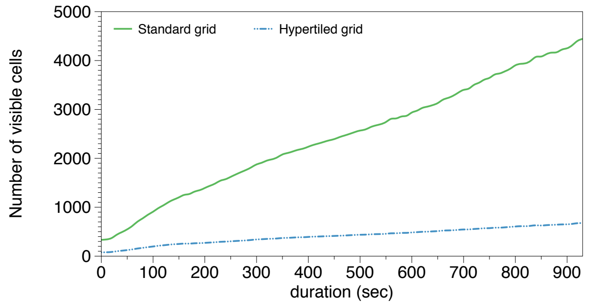

Figure 22. Starcraft 2 visible cells vs. hypergrid cells (10 second buckets)

With a straightforward implementation, Bob would need to encrypt all his units' data for each level of the hypergrid. But this is not necessary. For each non-empty hypergrid cell, Bob simply encrypts the keys for the grid cells containing those units. Furthermore, Bob's units are likely to be clustered together, so they will occupy relatively few hypergrid cells. Even though we increase the expense of units, hypergrids are almost always a worthwhile trade-off: players will have vision of more tiles than they have units in, and sending data for a unit is less expensive than querying o_k. The hypergrid technique works especially well if there are small areas of the map that provide great visibility, such as StarCraft II's Xel'naga towers (figure 23. By making the tower's radius a hypergrid cell, Alice will only need to send one query for all the territory revealed by the tower.

Figure 23. A Starcraft II Xel'naga watch tower screenshot

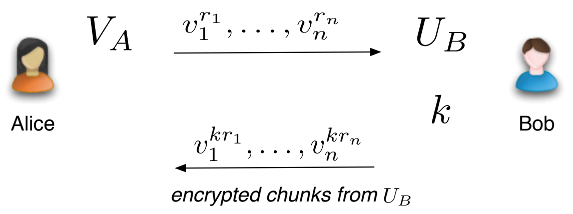
V_A
$v_1^{r_1}, \ldots, v_n^{r_n}$
U_B
Alice
k
Bob
$v_1^{kr_1}, \ldots, v_n^{kr_n}$
encrypted chunks from U_B

Figure 21. Computing $V_A \cap U_B$ where $V_A = \{v_1, \ldots, v_n\}$

F. Multiplayer

When there are more than two players, the cost of the protocol remains reasonable. The queries $H_1(v)^r$ are independent of the player being queried, so Alice can broadcast them to all players at once. Similarly, Bob's data sections are independent of the player who is querying, so Bob can also broadcast them. The only per-opponent work that Bob needs to do is to compute o_k, and the only per-opponent work that Alice needs to do is to unblind and interpret the results.

VIII. OpenConflict

We built a prototype of the system described in the previous section called OpenConflict. Our prototype does not yet incorporate all the optimizations described in the previous section, but it is enough to demonstrate that the technique is feasible in modern real-time games.

We first implemented OpenConflict using standard cryptographic components, and benchmarked it on a Core i5 660 dual-core hyperthreaded processor running at 3.33 GHz. With the standard NIST elliptic curves, a single exponentiation took on the order of a millisecond. With 200 visibility hypertiles and 150 units per player, each player would take approximately 750 milliseconds just for the exponentiations – completely unacceptable.

Many newer elliptic-curve implementations are available that reduce exponentiation times by an order of magnitude or more, which is beginning to become feasible. However, we realized that we do not need the security level that these implementations provide, and could reduce the running times by another factor of 5 or so by using smaller parameters. Taking a hint from Dan Bernstein's Curve25519 software [3], we chose to use the Montgomery curve

$$y^2 = x^3 + ax^2 + x \pmod p$$

where $a := 6366$ and $p := 2^{127} - 1$.

The cardinality of this curve is $16 \cdot q_{\text{curve}}$, where q_{curve} is a prime slightly larger than 2^{123}, and the cardinality of its twist is $16 \cdot q_{\text{twist}}$, where q_{twist} is a prime slightly smaller than 2^{123}.

Because p is a Mersenne prime, this curve supports a very efficient implementation; exponentiations on it take only 11-12µs. Because it is a Montgomery curve, exponentiations are computed using the x-coordinate only, which simplifies the implementation and saves bandwidth. Whether a point is on the curve or on the twist gives away information, so we decided arbitrarily to hash points to the twist. In order to implement the map from grid cells to points on the twist, we first map the points to a number $k \notin \{-1, 0, 1\}$ mod 2^{127-1}. We then compute

$$x := \frac{a}{k^2 - 1} \quad \text{and} \quad w := -a - x = \frac{-k^2 a}{k^2 - 1}$$

Note that $w^2 + aw + 1 = x^2 + ax + 1$, so that

$$\begin{aligned} &(x^3 + ax^2 + x)(w^3 + aw^2 + w) \\ &= xw(x^2 + ax + 1)^2 \\ &= -\left(\tfrac{ka(x^2+ax+1)}{k^2-1}\right)^2 \end{aligned}$$

Since -1 is not square mod p, either $x^3 + ax^2 + x$ or $w^3 + aw^2 + w$ must be square and the other non-square (neither one can be 0, because $x \neq 0$, $w \neq 0$ and $a^2 - 4$ is not square mod p). Therefore either x or w is on the curve, and the other is on the twist. Since this map is invertible with constant probability, it can be substituted for the random-oracle hash in the previous section without damaging the security proof. To avoid leaking information from the cofactor of 16, we multiply r, r^{-1} and k by 16 before using them. This results in uniformly random points on the prime-order subgroup of the twist. For the stream cipher and to generate random numbers, we used ChaCha/12 [2]. This cipher is extremely fast and adds negligibly to the runtime. For hashing, we used SHA256.

A. Security

OpenConflict cryptographic primitives need to remain secure for the duration of a game, which is about an hour. Consequenetly, low-security parameters are sufficient. OpenConflict uses elliptic curves defined over a field of size $2^{127} - 1$. To assess the viability of these weak parameters, we implemented a curve of approximately 2^{58} points over the Mersenne prime $2^{61} - 1$. This prime is the largest Mersenne prime where products require a single 64-by-64-bit multiplication.

The best known algorithms take $O(\sqrt{q})$ time to solve discrete logarithms, so this curve should be in reach. We implemented a simple multithreaded baby-step/giant-step cracker for this curve, and found that after precomputation, the median time to find discrete logs using our test machine was 12 seconds. This is clearly too weak, but it allows us to estimate cracking times for curves over $2^{89}-1$ and $2^{127}-1$ at 72 machine-days and 3,200 machine-years, respectively. Thus an attacker willing to devote 1,000 times as much compute power to the task will have at most a 1-in-28,000 chance of breaking the key within an hour. We are confident that even the wealthiest, most brazen cheater would not consider this a worthwhile attack, so that curves over $2^{127}-1$ are presently sufficient to keep game state secure for the duration of the game. Curves over $2^{89}-1$ would speed up OpenConflict by about 33%, but a cheater with a cluster of machines could could possibly break a key over the course of a game.

B. Measurements

We benchmarked the software on a Core i5 660 at 3.33 GHz, a dual-core hyperthreaded processor. We only used a single core, expecting that the other cores would be taken up by game logic; however, we observed that the code threads and hyper-threads very well, so a game which is willing to devote more cores to cryptography would see considerable speedups, and in particular reduced latency.

$v \downarrow \quad u \rightarrow$	100	200	300	400	500
100	5ms	6ms	8ms	9ms	11ms
200	8ms	9ms	11ms	12ms	14ms
300	11ms	13ms	14ms	16ms	17ms
400	14ms	16ms	17ms	19ms	20ms
500	17ms	19ms	20ms	22ms	24ms

Table III
OPENCONFLICT RUNTIMES

The benchmarks in table III show timings for the entire OpenConflict protocol for different numbers u of units and v of visible grid hypertiles. We see that units cost about 15µs, and visibility tiles cost about 30µs. This time is distributed across Alice and Bob's CPUs, so with careful pipelining the latency may be less than the numbers shown.

IX. DISCUSSION: PREVENTING ACTIVE ATTACKS

So far we have only considered defenses against passive eavesdropping attacks, which is a must in online game security. The next step is defending against active attacks, where a malicious player interferes with the in-memory state of the game to add units or resources. Preventing active attacks as they occur is quite difficult: all players would have to verify (in zero-knowledge) the actions of all other players. However, detecting active attacks after the game is over is far simpler.

One approach is as follows: during the game every client signs all its network traffic, and periodically (e.g. once every ten seconds) sends to all other players a signed commitment to (a hash of) all its actions that affected the state of the game. These commitments reveal nothing about the user's actions to the other users. At the end of the game, all users upload the commitments they received to a central server along with a cleartext trace of their actions during the game. The central server can verify that the transcript is consistent with the rules of the game and with the commitments generated during the game. An active attack would, by definition, show up as an inconsistency with the rules of the game at some point during the game. Since the state is signed, no player can frame another player and consequently, the central server knows exactly which player cheated.

When active cheating is detected after the game is over, the central server does not credit the "win" to the winning party. As a result, the winning player does not advance in the rankings and may be penalized for his actions. One difficulty with this approach is the randomness used by game clients during the game. Active attackers can control their client's random number generator and bias the game in their favor. One solution is to require each client to commit to a seed for a pseudorandom generator at the beginning of the game so that no more true random bits are generated during the game. The actual randomness used during the game is the xor of all the players' pseudorandom sequences, which is revealed as needed. At the end of the game the central server checks that each client contributed the correct pseudorandom bit at every step. Similarly, each client can commit to the seed of the pseudorandom generator it used in the oblivious set intersection protocol, and reveal the seed at the end of the game, so that the server can check that this has been done honestly. There are many challenges to overcome in implementing this high-level outline; we leave them as an area for fruitful future work.

X. ADDITIONAL RELATED WORK

Set intersection protocols. Generally speaking, set intersection protocols fall into two categories:

- The first approach makes use of an additively homomorphic encryption scheme and is based on evaluating polynomials given their encrypted coefficients [10], [20], [14], [5].
- The second approach makes use of Oblivious PRFs (OPRFs) [13], [17] or unpredictable functions [18], [8].

While we do not discuss these techniques in detail here, we mention that protocols in the second category tend to be more efficient and simpler than protocols in the first category. In this paper we used an adaption of the protocol due to Jarecki and Liu [18].

Game security. So far most of the work on improving game security focused on detecting bots. In [11] the authors show how to use machine learning algorithm to detect bots based on players sequence of action. In [9] the authors propose to use tamper-resistant hardware to detect cheaters. In [6] provide an empirical study of online game cheating. On the attack side, in [15], presents "The Supervisor, a kernel-level rootkit made specifically to bypass The Warden, Blizzard Entertainments anti-cheating technology. In the Defcon talk "So Many Ways to Slap A Yo-Ho: Xploiting Yoville and Facebook for Fun and Profit" [24] the authors showed how to cheat on the Yoville game. The most popular tool to tampers with game memory is called cheat engine [7]. This program only allows to manipulate memory block by block and do not offers any of the visualization techniques and advanced features provided by *Kartograph*. Many game hacking techniques are discussed in [16].

Visualization. Visualization techniques to reverse engineering binary and data files was successfully applied in[4]. Heatmap visualization where successfully applied to find RC4 anomaly in [23]

Securely outsourcing computations. Attesting to games is closely related to the problem of outsourcing computation to untrusted clients [22] which have been extensively studied for network clients [19] and more recently for Web applications [25].

XI. Conclusions

We presented two systems: an attack tool called *Kartograph* that is capable of lifting the fog of war in online games and a defense system called OpenConflict that prevents passive attacks by distributing game state among the players. OpenConflict is an elegant real-world application of private set intersection protocols. We showed that an optimized protocol is sufficiently fast to be used in a real game. In addition to the tools, we also performed an analysis of 1000 Starcraft II games to establish performance requirements for OpenConflict. Security in online games is a fruitful area of research. While this paper explored defenses against passive map hack attacks, much work remains in defending against active attacks.

Acknowledgments

The authors are supported by NSF, DARPA, and AFOSR.

References

[1] E. S. Association. Essential game facts 2009. http://www.theesa.com/facts/pdfs/ESA_Essential_Facts_2010.PDF, 2010.

[2] D. J. Bernstein. Chacha, a variant of salsa20. cr.yp.to/chacha.html.

[3] D. J. Bernstein. Curve25519: new diffie-hellman speed records. In *Proc. of PKC'06*, pages 207–228, 2006.

[4] G. Conti, E. Dean, M. Sinda, and B. Sangster. Visual reverse engineering of binary and data files. *Visualization for Computer Security*, pages 1–17, 2008.

[5] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *ACNS 2009*, volume 5536 of *LNCS*, pages 125–142, 2009.

[6] S. De Paoli and A. Kerr. The Cheating Assemblage in MMORPGs: Toward a sociotechnical description of cheating.

[7] C. engine team. Cheat engine. http://www.cheatengine.org/.

[8] A. Evfimievski, J. Gehrke, and R. Srikant. Limiting privacy breaches in privacy preserving data mining. In *PODS'03*, pages 211–222, 2003.

[9] W. Feng, E. Kaiser, and T. Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20. ACM, 2008.

[10] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Proc. of Eurocrypt 2004*, volume 3027 of *LNCS*, pages 1–19, 2004.

[11] S. Gianvecchio, Z. Wu, M. Xie, and H. Wang. Battle of Botcraft: fighting bots in online games with human observational proofs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 256–268. ACM, 2009.

[12] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proc. of STOC'87*, pages 218–229, 1987.

[13] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Proc. of TCC'08*, pages 155–175, 2008.

[14] C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *Proc. of PKC 2010*, volume 6056 of *LNCS*, pages 312–331, 2010.

[15] G. Hoglund. Hacking world of warcraft: An exercise in advanced rootkit design. In *Black Hat*, 2006.

[16] G. Hoglund and G. McGraw. Exploiting online games: cheating massively distributed systems. 2007.

[17] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In *Proc. of TCC'09*, pages 577–594, 2009.

[18] S. Jarecki and X. Liu. Fast secure computation of set intersection. In *Proc. of Security and Cryptography for Networks (SCN'10)*, volume 6280 of *LNCS*, pages 418–435, 2010.

[19] S. Jha, S. Katzenbeisser, and H. Veith. Enforcing semantic integrity on untrusted clients in networked virtual environments. 2007.

[20] L. Kissner and D. Song. Privacy-preserving set operations. In *Proc. of CRYPTO'05*, pages 241–257, 2005.

[21] J. Levin. The dark side of winsock. www.defcon.org/images/defcon-13/dc13-presentations/DC_13-Levin.pdf.

[22] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *NDSS*, 1999.

[23] D. Schweitzer and L. Baird. Discovering an RC4 anomaly through visualization. In *Proceedings of the 3rd international workshop on Visualization for computer security*, pages 91–94. ACM, 2006.

[24] S. M. W. to Slap A Yo-Ho:: Xploiting Yoville, F. for Fun, P. Strace, S. Barnum, EvilAdamSmith, Kanen, and J. Tyson. So many ways to slap a yo-ho:: Xploiting yoville and facebook for fun and profit. Defcon 18, 2010.

[25] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186. ACM, 2009.

[26] Wikipedia. The blizzard warden. http://en.wikipedia.org/wiki/Warden_(software).